How DOS Programs Can Use Over 1MB of RAM

BY JEFF PROSISE

When engineers at IBM designed the IBM PC in late 1980, few of them could have imagined that 1MB of RAM would soon seem like so little. 16K was a lot of RAM in those days, and personal computers with 64K of RAM were a luxury that few could afford.

Today it's hard to buy a PC with as little as 1MB of RAM. Most 386s come with at least 2MB, while a typical 486 comes with 4MB or more. Sadly, the majority of this RAM goes unused on most PCs unless they happen to be outfitted with Microsoft Windows, DESQview 386, or a similar protected-mode supervisory program, because few DOS applications are written to use more than 640K. Fortunately, this needn't be the case.

DOS is a real-mode operating system. That used to mean that the microprocessor was limited to addressing 1MB of RAM. However, today there are plenty of ways for programmers writing DOS applications to break the 1MB barrier without forsaking DOS. Here, in terms that nonprogrammers can understand, I'll explain how programmers use the Expanded Memory Specification (EMS), the Extended Memory Specification (XMS), DOS extenders, the LOADALL instruction, and increased segment sizes to break the 1MB limit.

EXPANDED MEMORY SPECIFICATION First and foremost among methods for circumventing the 1MB cap on real-mode memory is the Expanded Memory Specification. The first version of the EMS to gain wide acclaim, Version 3.2, was tendered by Lotus, Intel, and Microsoft in September 1985 (3.2 was actually the second version of the EMS; the first, released earlier in 1985, was, for reasons unexplained, named EMS 3.0). Version 4.0, the version most widely used today, followed in 1987.

The EMS describes the functional interface for a *bank-switched* memory scheme. EMS bank-switching works by reserving a range of memory addresses (usually 64K) and swapping blocks of EMS memory in and out of this address space. In the EMS world, the blocks are usually 16K in length and are called *pages.*

Figure 1 shows how EMS works. The rectangle on the left represents the 1MB of address space that the CPU can access directly. Conventional memory extends from 0K to 640K, and upper memory extends from 640K to 1MB. The rectangle on the right represents 2MB of expanded memory divided into 128 evenly spaced 16K pages. The key is the 64K space set aside in upper memory to serve as the EMS page frame. The page frame contains room for four 16K EMS pages. By placing calls to a driver called an *expanded memory manager* (EMM), an application program can map any arbitrary page of expanded memory to any of the four pages in the page frame. Expanded memory that is mapped to the page frame can be read and written just as if it were normal memory. By swapping EMS pages in and out of the page frame, a program can access every page of expanded memory it was allocated. Another way of looking at it is that the page frame serves as a window into expanded memory. You can't access all the memory at once, but you can access it all in turn by moving pages in and out of the window.

Using methods such as the Expanded Memory Specification, Extended Memory Specification, and DOS extenders, real-mode DOS programs can access up to 4GB of RAM.

An important point to note is that a program must be specially written to take advantage of expanded memory. It's up to the program to allocate expanded memory from the EMM (using an entirely different set of function calls from the ones that allocate DOS memory), to map pages of expanded memory to and from the page frame, and to copy data to and from those pages. That's why not all programs can use expanded memory.

Microsoft chairman Bill Gates once called the EMS a kludge. If you're thinking in terms of protected-mode memory, EMS *is* a kludge. But it's a workable one at that, and it's so easy to write to that it's a wonder more programs don't use it. Expanded memory can be added to any PC, even old 8086- and 8088-based machines. Although expanded memory boards aren't all that common anymore, expanded memory is available to almost everyone these days, because DOS 5.0 and 6.0 come with a driver (EMM386.EXE) that, if asked, converts extended memory—ubiquitous on 386 and 486 PCs—to EMS 4.0 expanded memory. Significantly, DOS 6.0's EMM386.EXE driver allocates EMS and XMS memory from a shared pool, so you don't have to decide ahead of time whether to configure the extended memory in your system as EMS or XMS memory; EMM386.EXE will dish it out as EMS memory to one program and as XMS memory to the next.

EXTENDED MEMORY SPECIFICATION Whi[illegible]expanded memory may not be all tha[illegible] [illegible]mmon anymore, extended memory is. Extended memory is all the RAM above 1MB on 286, 386, and 486 PCs. On the surface, an application program that wants to use extended memory for data storage must go to great lengths to do so. First it has to switch to protected mode. Then it does whatever it needs to with the data in extended memory, perhaps copying it to conventional memory where it can be manipulated more easily. Then it must switch back to real mode. It doesn't sound too hard, but in practice it involves a lot of hard work and requires a thorough understanding of the CPU's operation. To make matters worse, DOS doesn't manage extended memory as it does conventional memory, so it's difficult for a program to tell when the extended memory it desires is being used by another program.

Enter the Extended Memory Specification, the product of a collaborative effort between Microsoft, Intel, Lotus, and AST Research. Introduced in August 1988, the XMS defines a software interfa[illegible] extended memory. Programs that seek to use extended memory for data storage may do so by invoking the services of an extended memory manager (XMM). The XMM takes care of the nuances associated with switching to protected mode and back; to the application program, all that's required to copy data to or from extended memory is a simple function call or two. Some XMM implementations don't even bother to switch to protected mode. Instead they exploit hidden features of the CPU's design (features such as the undocumented LOADALL instruction, which was discussed in the June 16, 1992, Tutor column) that permit the CPU to reach into extended memory from real mode. The bottom line? On a PC with an XMM such as HIMEM.SYS installed, it is simple—almost trivial—for programmers to write applications that store information in extended memory. And since the XMM keeps track of who owns what in extended memory, programs need not fear overwriting blocks of memory allocated t[illegible]ther application programs.

[illegible]he XMS does more than manage extended memory. It also provides a device-independent means for programs to access the 65,520 bytes of extended memory called the HMA (high memory area). The HMA is special because it, unlike the rest of extended memory, is easily accessed from real mode using standard segment:offset addressing conventions once the microprocessor's A20 address line is enabled. The XMS provides convenient functions for programs to enable and disable the A20 address line. The XMS also defines a software interface for programs to allocate and deallocate upper memory blocks created by 386 memory managers such as EMM386.EXE.

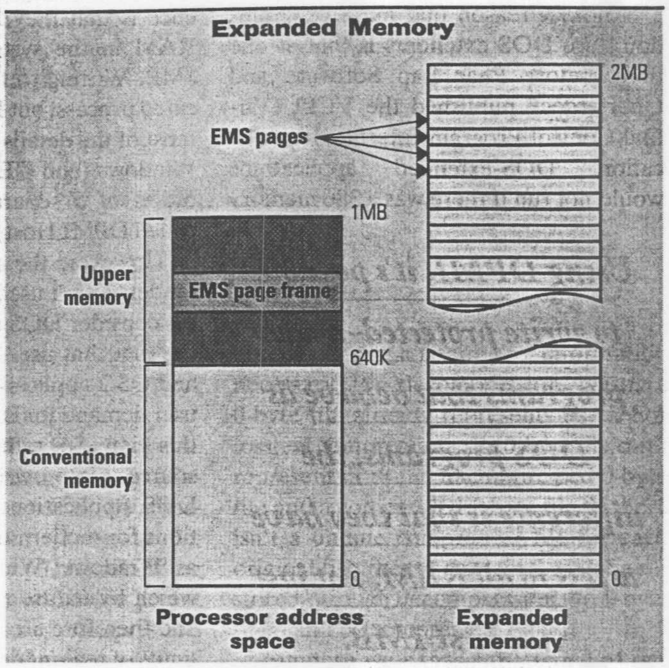

Figure 1: **To access data in expanded memory, a program places a call to the expanded memory manager to swap 16K pages of EMS memory (shown at right) into the page frame in upper memory (left).**

Extended memory placed under the control of an XMM is often referred to as *XMS memory*. Like EMS memory, XMS memory isn't doled out to programs automatically; to use it, a program must want to use it and then take special steps to do so. Therefore not all programs use XMS memory. But the number of programs that do is increasing every day.

DOS EXTENDERS DOS extenders are products that permit developers to transform real-mode application programs into full-fledged protected-mode application programs that enjoy access to extended memory. Examples of applications built with DOS extenders include AutoCAD, Releases 11 and 12, Mathematica, and Lotus 1-2-3, Release 3.*x*. To use these programs, all you have to do is load them up on a 386 or 486 with gobs of RAM, and then let them go. The programs do the rest.

More specifically, the DOS extenders do the rest. When a DOS-extended program is started, the DOS extender, which is bound into the program, takes control and switches the CPU to protected mode. Then it loads the application program that it was bound to. When the program places calls to real-mode DOS and BIOS services, the DOS extender either handles those calls itself (in protected mode) or temporarily switches back to real mode, allows DOS or the BIOS to process the service request, and then switches once more to protected mode. This simple description belies the amount of work that must go on behind the scenes to accomplish this little feat of magic, but the beauty of it is that the programmer needn't be concerned with any of it. The programmer can write to an application program interface he's familiar with (the DOS API), abide by a few additional rules imposed by protected-mode operation, and lo and behold, an application that's not constrained by the 1MB barrier is born.

So why don't more programs use DOS extenders? That's a good question. One reason for this could be that many programmers perceive DOS extenders as complex pieces of software that have a steep learning curve, so they put off using them. This is too bad, since an experienced programmer, armed with the proper software tools, can be writing DOS-extended applications in a matter of weeks—maybe even days.

Another reason that more programs don't use DOS extenders is that at one time, before Phar Lap Software and Quarterdeck published the VCPI (Virtual Control Program Interface) specification, DOS-extended applications would not run if there was a 386 memory manager such as QEMM-386 active in the system. Conflicts occurred because both programs (the DOS extender and the memory manager) performed processor mode switches and tried to claim all the extended memory in the system. VCPI resolved those conflicts by defining a communication protocol allowing the DOS extender to borrow RAM from the memory manager and to use services built into the memory manager to switch processor modes. DOS 6.0's EMM-386.EXE fully supports VCPI and therefore supports DOS-extended programs; popular third-party memory managers such as QEMM-386 and 386MAX also support VCPI.

Using DPMI, it's possible to write protected-mode programs that behave as DOS programs; the difference is that they have access to all RAM, not just the first 1MB.

The DOS Protected-Mode Interface (DPMI) has succeeded VCPI as the specification of choice to use when developing standalone protected-mode applications. DPMI defines a subset of DOS and BIOS function calls that may be used by protected-mode DOS programs, and it offers additional functions that DOS lacks for performing tasks such as allocating protected-mode memory, modifying descriptors (values used to translate virtual addresses to physical addresses in protected mode), and making calls to real-mode device drivers. Using DPMI, it is possible to write protected-mode programs that behave in most respects as if they were DOS programs; the difference is that they have access to all the RAM in the system, not just the first 1MB. Writing to the DPMI is a complicated process, but DOS extenders handle most of the details for you. 386MAX and Windows (and QEMM-386, with the addition of a separate software package called DPMI Host) support this spec.

However, the real reason DOS extenders aren't used more is that developers consider DOS a dying platform. And the fact that user demand for Windows and OS/2 applications is far greater than user demand for DOS programs supports this view. So, rather than pour their resources into writing protected-mode DOS applications, they write applications for modern operating systems such as Windows, Windows NT, and OS/2, which by nature run in protected mode and therefore are not constrained by the limits of real-mode memory.

ACCESSING 4GB FROM REAL MODE Earlier I mentioned the undocumented LOADALL instruction, which provides a back-door mechanism for programmers to access the entire protected-mode address space without leaving real mode. LOADALL loads all the CPU's registers, including the descriptor cache registers used to form higher-than-1MB addresses in protected mode, from a table in low memory. By loading the descriptor cache registers with the appropriate values with LOADALL and then using the 32-bit index registers found on the 386 and 486 to specify offset addresses, a program can access any part of memory from real mode. The LOADALL instruction was used in DOS as early as Version 3.3 and was also used in the 286 versions of OS/2.

Recently, another method of accessing 4GB (or 4,000MB) of memory (the physical addressing limits of the 386 and 486) was brought to my attention by a reader on PC MagNet. Here's how it works:

In protected mode, the size and location of a segment of memory is defined by base address and limit fields in the segment descriptor. In real mode, the base address of the segment is derived directly from the value in the segment register, and it too has a limit. When a 386 or 486 CPU is powered up in real mode, the segment limit is set to 64K so that the segment registers will work in real mode just like segment registers on an 8086. If we could somehow increase that limit, then we could access up to 4GB of RAM in real mode by using 32-bit registers to specify offsets from the base of the segment.

As you might have guessed, there is a way to change the limit. If you switch to protected mode, increase the limit, and switch back to real mode, the new limit stays put. For this reason, Intel recommends that progammers reset the segment registers to descriptors that have a 64K limit.

The mechanics of switching to protected mode, increasing the segment sizes associated with one or more segment registers, and switching back to real mode are not prohibitively difficult. If you'd like to see them implemented in code, read Chapter 18 of Al Williams's excellent book, *DOS 5: A Developer's Guide* (1991, M&T Publishing). In it, Williams presents the C and assembly language source codes for a program that increases the real-mode segment limits.

The downfall of these last two methods for reaching into extended memory from real mode is that they only concern the mechanics of accessing the memory; they do not take into account memory management issues such as allocating and deallocating blocks of memory in a cooperative manner.

ALL IN ALL... The problem with all the memory standards discussed here is that none provides seamless access to more than 1MB of RAM. If access were seamless, a program would be able to allocate the memory and then use it with standard DOS API functions. Instead a program has to go out of its way to access extended or expanded memory. That's where operating environments such as Windows and OS/2 are superior: Their free memory pools include all the memory in the system, not just RAM below 1MB.

DOS is not the operating system that we'll want to be using 10 years from now. But don't buy into the notion that DOS programs can't possibly access more than 640K of memory. They can, and in the opinion of most DOS users, it's high time we saw more DOS applications that take advantage of EMS, XMS, and other memory-extending techniques. □